

# Constructing the Bijective BWT

HIDEO BANNAI<sup>1,a)</sup> JUHA KÄRKKÄINEN<sup>3,b)</sup> DOMINIK KÖPPL<sup>1,2,c)</sup>  
 MARCIN PIĄTKOWSKI<sup>4,d)</sup>

**Abstract:** The Burrows-Wheeler transform (BWT) is a permutation whose applications are prevalent in data compression and text indexing. The *bijective BWT* (BBWT) is a bijective variant of it. Although it is known that the BWT can be constructed in linear time for integer alphabets by using a linear time suffix array construction algorithm, it was up to now only conjectured that the BBWT can also be constructed in linear time. We confirm this conjecture by proposing a construction algorithm that is based on SAIS, improving the best known result of  $\mathcal{O}(n \lg n / \lg \lg n)$  time to linear.

## 1. Introduction

The Burrows-Wheeler transform (BWT) [4] is a transformation permuting all symbols of a given string  $T\$$ , where  $\$$  is a symbol that is strictly smaller than all symbols occurring in  $T$ . The  $i$ -th entry of the BWT of  $T\$$  is the character preceding the  $i$ -th lexicographically smallest suffix of  $T\$$ , or  $\$$  if this suffix is  $T\$$  itself. Strictly speaking, the BWT is not a bijection since its output contains  $\$$  at an arbitrary position while it requests the input  $T$  to have  $\$$  as a delimiter symbol at its end in order to restore  $T$ . A variant, called the bijective BWT [15], is a *bijective* transformation, which does not require the artificial delimiter  $\$$ . It is based on the Lyndon factorization [5] of  $T$ . In this variant, the output consists of the last symbols of the lexicographically sorted cyclic rotations of all factors composing the Lyndon factorization of  $T$ .

In the following, we call the BWT *traditional* to ease the distinguishability of both transformations. It is well known that the traditional BWT has many applications in data compression [1] and text indexing [8–10]. Recently, such a text index was adapted to work with the bijective BWT [2].

**Related Work.** In what follows, we focus on a text  $T$  of length  $n$  whose characters are drawn from an integer alphabet. Thanks to linear time suffix array construction algorithms [14, 18], we can construct the traditional BWT based on the relation  $\text{BWT}[i] = T[\text{SA}[i] - 1]$  in linear time. Considering the bijective BWT, Gil and Scott [11] postulated that it can be built in linear time, but did not give

a construction algorithm. It is clear that the time is upper bounded by the total length of all conjugates [17, after Example 9], which is  $\mathcal{O}(n^2)$ . Mantaci et al. [17] also introduced the *extended* BWT, a generalization of the BBWT in that it is a BWT based on a set  $\mathcal{S}$  of *primitive* strings, i.e., strings that are not periodic. Hon et al. [12] provided an algorithm building the extended BWT in  $\mathcal{O}(n \lg n)$  time. Their idea is to construct the *circular suffix array*  $\text{SA}_\circ$  such that the  $i$ -th position of the extended BWT is given by  $T[\text{SA}_\circ[i] - 1]$ , where  $T$  is the concatenation of all strings in  $\mathcal{S}$ . Bonomo et al. [3] presented the most recent algorithm building the bijective BWT online in  $\mathcal{O}(n \lg n / \lg \lg n)$  time. In [3, Sect. 6], they also gave a linear time reduction from computing the extended BWT to computing the BBWT. Knowing that an irreducible word has exactly one conjugate being a Lyndon word, the reduction is done by exchanging each element of the set of irreducible strings  $\mathcal{S}$  by the conjugate being a Lyndon word, and concatenating these Lyndon words after sorting them in descending order. Consequently, a linear-time BBWT construction algorithm can be used to compute the extended BWT in linear time.

**Our Result.** In this article, we present a linear time algorithm computing the BBWT. The main idea is to adapt the suffix array construction algorithm SAIS [18] to compute the circular suffix array of the Lyndon factors. We obtain linear running time by exploiting some facts based on the nature of the Lyndon factorization.

## 2. Preliminaries

Our computational model is the word RAM model with word size  $\Omega(\lg n)$ . Accessing a word costs  $\mathcal{O}(1)$  time. In this article, we study strings on an *integer* alphabet  $\Sigma$  with size  $\sigma = n^{\mathcal{O}(1)}$ :

<sup>1</sup> Department of Informatics, Kyushu University, Japan

<sup>2</sup> Japan Society for Promotion of Science

<sup>3</sup> Helsinki Institute of Information Technology (HIIT), Finland

<sup>4</sup> Nicolaus Copernicus University, Toruń, Poland

a) bannai@inf.kyushu-u.ac.jp

b) juha.karkkainen@cs.helsinki.fi

c) dominik.koeppl@inf.kyushu-u.ac.jp

d) marcin.piatkowski@mat.umk.pl



- $R[i..|R|]$  is type **S** otherwise, i.e.,  $R[i..|R|] \prec_{\text{lex}} R[i + 1..|R|]$ .

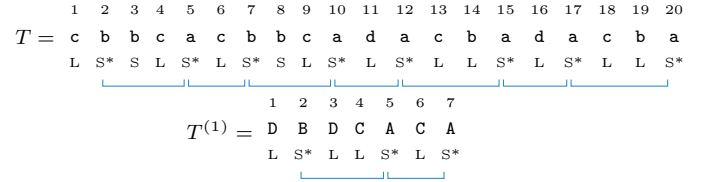
Since it is not possible that  $R[i..|R|] = R[i + 1..|R|]$ , SAIS assigns each suffix a type. An **S** suffix  $R[i..|R|]$  is additionally type **S\*** if  $i = 1$  or  $R[i - 1..|R|]$  is type **L**. The substring between two succeeding **S\*** suffixes is called an *LMS substring*. In other words,  $R[i..j]$  is an LMS substring if and only if  $R[i..|R|]$  and  $R[j..|R|]$  are type **S\*** and there is no  $k \in (i..j)$  such that  $R[k..|R|]$  is type **S\***. The types for the suffixes of our running example are given in Fig. 2.

Next, SAIS gives the LMS substrings a rank based on the *substring order* [18, Def. 3.3]: Given two LMS substrings  $S$  and  $U$  with  $S \neq U$ , we write  $S \prec_{\text{LMS}} U$  if and only if (a)  $S[i] < U[i]$  or (b)  $S[i]$  is type **L** and  $U[i]$  is type **S** or **S\***, for the smallest position  $i$  where (a)  $S[i] \neq U[i]$  or (b) the types of  $S[i]$  and  $U[i]$  differ. This order is on the left side of Fig. 3 for the LMS suffixes of the left side of Fig. 2.

Having the  $\prec_{\text{LMS}}$ -order of all LMS substrings, we can assign each LMS substring its  $\prec_{\text{LMS}}$ -rank, and replace the LMS substrings in  $R$  by the respective ranks, keeping the last character during a replacement remaining if this character is the first character of the subsequent LMS substring. See the right side of Fig. 2 for our running example. We recursively call SAIS on this text of ranks until all LMS substrings have a different rank, since then these ranks determine the order of the **S\*** suffixes of  $R$ . The order of the **S\*** suffixes of our running example are given in Fig. 3 on the right side. Having the order of the **S\*** suffixes, we allocate space for the suffix array, and divide the suffix array into buckets, grouping each suffix with the same starting character and same type (either **L** or **S**) into one bucket. Putting **S\*** suffixes in their respective buckets according to their order (smallest elements are the leftmost elements in the buckets), we can induce the **L** suffixes, as these precede either **L** or **S\*** suffixes. Since an **L** suffix immediately preceding an **S\*** suffix is smaller than an **L** suffix immediately preceding two or more **L** suffixes, we can induce all **L** suffixes by a scan of the suffix array from left to right: When accessing the entry  $\text{SA}_o[k] = i$ , write  $i - 1$  to the **L** type bucket with the character  $R[i - 1]$  if  $R[i - 1..|R|]$  is type **L**. Finally, we can induce those **S** suffixes that are not type **S\*** by scanning the suffix array from right to left: When accessing the entry  $\text{SA}_o[k] = i$ , write  $i - 1$  to the **S** type bucket with the character  $R[i - 1]$  if  $R[i - 1..|R|]$  is type **S**. We conduct these steps for our running example in Fig. 4.

In total, the induction takes  $\mathcal{O}(|R|)$  time. The recursion step takes also  $\mathcal{O}(|R|)$  time since there are at most  $|R|/2$  LMS substrings (there are no two text position  $R[i]$  and  $R[i + 1]$  with type **S\*** for  $i \in [1..n - 1]$ ).

However, with SAIS we cannot obtain  $\text{SA}_o$  ad-hoc, since we need to exchange  $\prec_{\text{lex}}$  with  $\prec_{\omega}$ . Although these orders are the same for Lyndon words, they differ for LMS substrings as can be seen in Fig. 5. Hence, we need to come up with an idea to modify SAIS in such way to compute  $\text{SA}_o$ .



**Fig. 2** Splitting  $T$  and  $T^{(1)}$  into LMS substrings. The rectangular brackets below the types represent the LMS substrings.  $T^{(1)}$  is  $T$  after the replacement of its LMS substrings with their corresponding ranks defined in Sect. 3.3 and on the left of Fig. 3.

LMS Substring	Contents	Non-Terminal
$T[2..5]$	bbca	D
$T[5..7]$	acb	B
$T[7..10]$	bbca	D
$T[10..12]$	ada	C
$T[12..15]$	acba	A
$T[15..17]$	ada	C
$T[17..20]$	acba	A

S* Suffix	Contents
$T[20]$	a
$T[17..20]$	acba
$T[12..20]$	acbadacba
$T[5..20]$	acbbcadacbadacba
$T[15..20]$	adacba
$T[10..20]$	adacbadacba
$T[2..20]$	bbcacbbcadacbadacba
$T[7..20]$	bbcadacbadacba

**Fig. 3** Ranking of the LMS substrings and the **S\*** suffixes of the text  $T$  given in Sect. 3.3 and Fig. 2. *Top*: LMS substrings assigned with non-terminals reflecting their corresponding rank in  $\prec_{\text{LMS}}$ -order. *Bottom*: **S\*** suffixes of  $T$  sorted in  $\prec_{\text{lex}}$ -order. Note that  $T[5..7] = \text{acb} \prec_{\text{lex}} \text{acba} = T[12..15] = T[17..20]$ , but  $\text{acba} \prec_{\text{LMS}} \text{acb}$ .

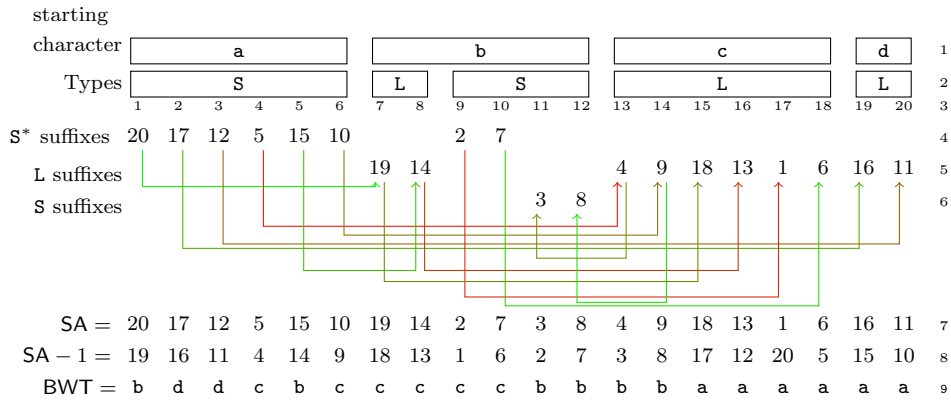
### 3.2 Our Adaptation

We want SAIS to sort Lyndon conjugates in  $\prec_{\omega}$ -order instead of suffixes in  $\prec_{\text{lex}}$ -order. For that, we introduce the notion of inf-suffixes, replacing the suffixes as the elements to sort in SAIS: Let  $R[i..]$  denote the infinite string  $R[i..e(T_x)]T_xT_x \cdots = \text{conj}_{i-1}(T_x)\text{conj}_{i-1}(T_x) \cdots$  with  $x$  such that  $i \in [\mathbf{b}(T_x)..e(T_x)]$ . We say that  $R[i..]$  is an *inf-suffix*. The *factorization borders* are between  $R[e(F_x)]R[\mathbf{b}(F_{x+1})]$  for  $x \in [1..t - 1]$ . Like in SAIS, we distinguish between **L** and **S** inf-suffixes:

- $R[i..]$  is type **L** if  $R[i..] \succ_{\text{lex}} R[j..]$ , and
- $R[i..]$  is type **S** if  $R[i..] \prec_{\text{lex}} R[j..]$ ,

where  $j := (i - \mathbf{b}(T_x) + 1) \bmod |T_x| + \mathbf{b}(T_x)$  is either  $i + 1$  or  $\mathbf{b}(T_x)$  if  $i = e(T_x)$ , and  $x$  is given such that  $i \in [\mathbf{b}(T_x)..e(T_x)]$ . When speaking about types, we do not distinguish between an inf-suffix and its starting position in  $R$ . This definition assigns all positions of  $R$  a type except those belonging to a Lyndon factor of length one. In all other cases, thanks to the Lyndon factorization, this definition matches the definitions of **L** and **S** suffixes of the SAIS algorithm. That is because of two facts:

- A Lyndon factor  $T_x$  of length at least two starts with the smallest character among all other characters of  $T_x$ . Since a Lyndon word is border-free,  $R[\mathbf{b}(T_x)]$  is type **S**.
- Due to the Lyndon factorization,  $R[\mathbf{b}(T_x)..|R|] \succ_{\text{lex}} R[\mathbf{b}(T_{x+1})..|R|]$  for  $x \in [1..t - 1]$ . Hence, the suffix



**Fig. 4** Inducing L and S suffixes from the  $\prec_{lex}$ -order of the  $S^*$  suffixes given in Fig. 2. Rows 1 and 2 show the partitioning of SA into buckets, first divided by the starting characters of the respective suffixes, and second by the types L and S. Row 4 is SA after inserting the  $S^*$  suffixes according to their  $\prec_{lex}$ -order rank obtained from the right of Fig. 3. The  $S^*$  (resp. L) suffixes induce the L (resp. S) suffixes in Row 5 (resp. Row 6). Putting all together yields SA in Row 7. In the penultimate row SA - 1, each text position stored in SA is decremented by one. The last row shows  $T[(SA - 1)[i]] = BWT[i]$  in its  $i$ -th column, which is the BWT of  $T$ . This BWT is not reversible since the input is not terminated with a unique character like \$. To obtain the BWT of  $T\$$ , we first write  $T[SA[1]] = T[20] = a$ , and then BWT, but exchanging  $BWT[SA^{-1}[1]] = BWT[17] = a$  with \$, i.e.,  $abddcbccccbbbaa\$aaa$ .

$U$	$V$	$\prec_{lex}$	$\prec_{\omega}$	$\prec_{LMS}$
aba	aca	<	<	<
adc	adcb	<	>	>
acb	acba	<	<	>

**Fig. 5** Comparison of the three orders studied in this paper applied to LMS substrings. Assume that  $U$  and  $V$  are substrings of the text, neighbored by a character  $d$  such that the first and the last character of both  $U$  and  $V$  start with an  $S^*$  suffix. We mark with the signs  $<$  and  $>$  whether  $U$  is smaller or respectively larger than  $V$  according to the corresponding order. The orders can differ only when one string is the prefix of another string, as this is the case in the last two rows.

$R[e(T_x)..|R|]$  starting at  $R[e(T_x)]$  has to be lexicographically larger than the suffix  $R[e(T_x) + 1..|R|]$ , otherwise we could extend the Lyndon factor  $T_x$ .

To give all positions a type, we introduce the type  $S^*$ , which is handled like a special case of type S:

- If  $R[i..]$  is type S, it is further type  $S^*$  if  $R[j..]$  is type L with  $j := (|T_x| + i - b(T_x) - 1) \bmod |T_x| + b(T_x)$  being either  $i - 1$  or  $e(T_x)$  if  $i = b(T_x)$ .
- $R[b(T_x)]$  is type  $S^*$  for every  $x \in [1..t]$ .

If  $T_x$  and  $T_{x+1}$  are longer than one, then the types of all positions of  $R[b(T_x) + 1..e(T_{x+1})]$  match the original SAIS types.\*1 That is because the second condition comes into play only in the case that  $|T_x| = 1$ , since otherwise the last character  $R[e(T_x)]$  with  $R[e(T_x)] > R[b(T_x)]$  is type L. Further, since  $R[e(T_x)..|R|] \succ_{lex} R[b(T_x)..|R|] \succ_{lex} R[b(T_{x+1})..|R|]$ , the suffix  $R[e(T_x)..|R|]$  is an L suffix.

Next, we define the equivalent to the LMS substrings for the inf-suffixes, which we call *LMS inf-suffixes*. We want the LMS inf-suffixes to be contained inside the Lyndon factors since the  $\prec_{\omega}$ -order of a conjugate depends only on the order of its characters, and not, unlike suffixes, on all suc-

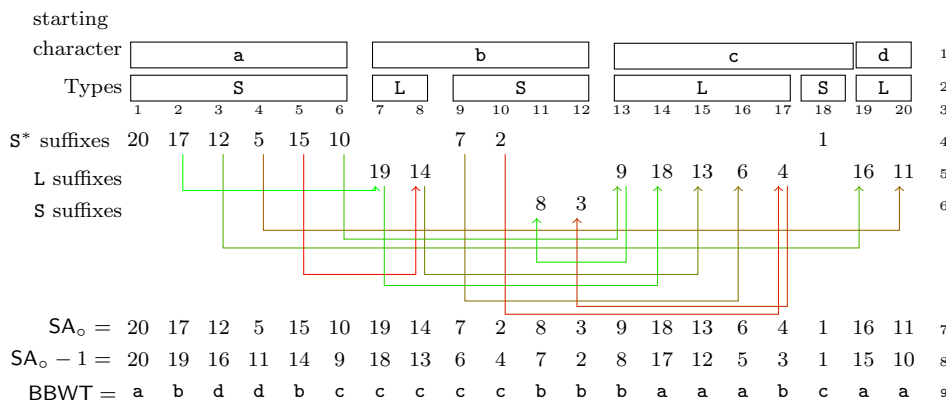
\*1 If  $T[b(T_{x-1})] = T[b(T_x)]$  and  $|T_{x-1}| = 1$ , then  $T[b(T_x)..|R|]$  is not an  $S^*$  suffix by the original definition.

ceeding characters in the text. To obtain this property, we only have to change the original definition of the LMS substrings slightly: Stipulating that  $T_x[|T_x| + 1] = T_x[1]$ , for  $1 \leq i < j \leq |T_x| + 1$ , the substring  $T_x[i..j]$  is an *LMS inf-substring* if  $T_x[i]$  and  $T_x[j]$  are type  $S^*$  and there is no  $k \in (i..j)$  such that  $T_x[k]$  is type  $S^*$ . This definition differs from the original LMS substrings only for the last LMS inf-substring of each Lyndon factor. Here, we append  $T_x[1]$  instead of  $T_{x+1}[1]$  to the suffix starting with the last type  $S^*$  position of  $T_x$ . If  $T_x$  has length one, it is possible that  $R[b(T_x)..|R|]$  is not an  $S^*$  suffix, while  $R[b(T_x)..]$  is always an  $S^*$  inf-suffix with the associated LMS inf-substring  $R[b(T_x)]R[b(T_x)]$ .

Exactly as in the SAIS recursion step, we map the LMS inf-substrings to meta-characters having its  $\prec_{LMS}$ -rank assigned. Since the Lyndon factorization of the string based on the meta-characters has the same factorization borders as the original string, we can continue with our deviation of SAIS by building LMS inf-substrings of the text based on the meta-characters while keeping the same factorization borders.

By doing so, we compute the  $\prec_{\omega}$ -order of all conjugates of the Lyndon factors of  $R$  (instead of the lexicographic order of all suffixes of  $R$ ). The correctness follows by construction: Instead of partitioning the suffixes into LMS substrings, we partition the Lyndon factors whose factorization borders happens to coincide with some borders of the LMS substrings. We use the same trick of the LMS substring partitioning, since we can obtain the  $\prec_{lex}$  of the  $S^*$  inf-suffixes from the  $\prec_{lex}$ -order of the LMS inf-substrings in the same way as obtaining the  $\prec_{lex}$  of the  $S^*$  suffixes from the  $\prec_{lex}$ -order of the LMS substrings. Finally, the induction steps can be conducted in the same way as in SAIS when taking care of the Lyndon boundaries, i.e., moving to the end of a





**Fig. 8** Inducing L and S inf-suffixes from the  $\prec_{lex}$ -order of the  $S^*$  inf-suffixes given in Fig. 6. Rows 1 and 2 show the partitioning of  $SA_o$  into buckets, first divided by the starting characters of the respective inf-suffixes, and second by the types L and S. Row 4 is  $SA_o$  after inserting the  $S^*$  inf-suffixes according to their  $\prec_{lex}$ -rank obtained from the right of Fig. 7. The  $S^*$  (resp. L) inf-suffixes induce the L (resp. S) inf-suffixes in Row 5 (resp. Row 6). Putting all together yields  $SA_o$  in Row 7. In the penultimate row  $SA_o - 1$ , each text position stored in  $SA_o$  is decremented by one, wrapping around a Lyndon factor if necessary (for instance,  $(SA_o - 1)[2] = 19 = e(T_5)$  since  $SA_o[2] = 17 = b(T_5)$ ). The last row shows  $T[(SA_o - 1)[i]]$  in its  $i$ -th column, which is the BBT of  $T$  as given in Fig. 1.

References

[1] D. Adjeroh, T. Bell, and A. Mukherjee. *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*. Springer, 2008.

[2] H. Bannai, J. Kärkkäinen, D. Köppl, and M. Patkowskii. Indexing the bijective BWT. In *Proc. CPM*, volume 128 of *LIPICs*, pages 17:1–17:14, 2019.

[3] S. Bonomo, S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino. Sorting conjugates and suffixes of words in a multiset. *Int. J. Found. Comput. Sci.*, 25(8):1161, 2014.

[4] M. Burrows and D. J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994.

[5] K. T. Chen, R. H. Fox, and R. C. Lyndon. Free differential calculus, IV. The quotient groups of the lower central series. *Annals of Mathematics*, pages 81–95, 1958.

[6] F. Dolce, A. Restivo, and C. Reutenauer. On generalized Lyndon words. *Theor. Comput. Sci.*, 777: 232–242, 2019.

[7] J. Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983.

[8] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. FOCS*, pages 390–398, 2000.

[9] P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.

[10] T. Gagie, G. Navarro, and N. Prezza. Optimal-time text indexing in BWT-runs bounded space. In *Proc. SODA*, pages 1459–1477, 2018.

[11] J. Y. Gil and D. A. Scott. A bijective string sorting transform. *ArXiv 1201.3077*, 2012.

[12] W. Hon, C. Lu, R. Shah, and S. V. Thankachan. Succinct indexes for circular patterns. In *Proc. ISAAC*, volume 7074 of *LNCS*, pages 673–682, 2011.

[13] W. Hon, T. Ku, C. Lu, R. Shah, and S. V. Thankachan. Efficient algorithm for circular Burrows-Wheeler transform. In *Proc. CPM*, volume 7354 of *LNCS*, pages 257–268, 2012.

[14] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006.

[15] M. Kuffleitner. On bijective variants of the Burrows-Wheeler transform. In *Proc. PSC*, pages 65–79, 2009.

[16] R. C. Lyndon. On Burnside’s problem. *Transactions of the American Mathematical Society*, 77(2): 202–215, 1954.

[17] S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino. An extension of the Burrows-Wheeler transform. *Theor. Comput. Sci.*, 387(3):298–312, 2007.

[18] G. Nong, S. Zhang, and W. H. Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Trans. Computers*, 60(10):1471–1484, 2011.